

WHAT IS CLAIMED IS:

- 1 1. A method of providing storage reclamation in a multiprocessor computer
2 system, the method comprising:
3 maintaining respective reference counts for shared objects;
4 accessing pointers to the shared objects using lock-free pointer operations to
5 coordinate modification of respective reference counts;
6 freeing storage associated with a particular one of the shared objects only once
7 the corresponding reference count indicates that the particular shared
8 object is unreferenced.

- 1 2. The method of claim 1, wherein the lock-free pointer operations ensure
2 that:
3 if a number of pointers referencing the particular shared object is non-zero,
4 then so too is the corresponding reference count; and
5 if no pointers reference the particular shared object, then the corresponding
6 reference count eventually becomes zero.

- 1 3. The method of claim 2,
2 wherein at any given instant, a number of pointers to the particular shared
3 object may differ from the corresponding reference count.

- 1 4. The method of claim 1, wherein the lock-free pointer operations include a
2 load operation that loads a shared pointer value to a local pointer variable and
3 employs:
4 a double-compare-and-swap (DCAS) primitive to increment a reference count
5 of a first shared object, if any, referenced by the shared pointer value
6 while ensuring continued existence thereof; and
7 a compare-and-swap (CAS) primitive to decrement a reference count of a
8 second shared object, if any, referenced by a pre-load value of the local
9 pointer variable.

1 5. The method of claim 1, whercin the pointer operations include a store
2 operation that stores a local pointer value to a shared pointer variable and employs:
3 a compare-and-swap (CAS) primitive to increment a reference count of a first
4 shared object, if any, referenced by the local pointer value;
5 a compare-and-swap (CAS) primitive to update the shared pointer variable
6 with the local pointer value; and
7 a compare-and-swap (CAS) primitive to decrement a reference count of a
8 second shared object, if any, referenced by a pre-store value of the
9 shared pointer variable.

1 6. The method of claim 1, wherein the pointer operations include a copy
2 operation that copies a local pointer value to a local pointer variable and employs:
3 a compare-and-swap (CAS) primitive to increment a reference count of a first
4 shared object, if any, referenced by the local pointer value; and
5 a compare-and-swap (CAS) primitive to decrement a reference count of a
6 second shared object, if any, referenced by a pre-copy value of the
7 local pointer variable.

1 7. The method of claim 1, wherein the pointer operations include a destroy
2 operation that:
3 decrements a reference count of a shared object identified by a supplied
4 pointer value; and
5 frees the identified shared object if the corresponding reference count has
6 reached zero.

1 8. The method of claim 7,
2 wherein, prior to the freeing, the destroy operation recursively follows pointers
3 defined in the shared object if the corresponding reference count has
4 reached zero.

1 9. The method of claim 1, employed in access operations on a composite
2 shared object that includes zero or more of the shared objects.

1 10. The method of claim 9,
2 wherein the composite shared object is embodied as a double ended queue
3 (deque);
4 wherein the shared objects include nodes of the deque; and
5 wherein the access operations implement push and pop accesses at opposing
6 ends of the deque.

1 11. The method of claim 10, wherein the push accesses employ:
2 a pair of compare-and-swap (CAS) primitives to increment a reference count
3 of a pushed node;
4 a double compare-and-swap (DCAS) primitive to splice the pushed node onto
5 the deque while mediating competing accesses to the deque;
6 a pair of compare-and-swap (CAS) primitives to decrement the reference
7 count of respective shared objects, if any, referenced by overwritten
8 pre-splice pointer values.

1 12. The method of claim 11, wherein the pairs of compare-and-swap (CAS)
2 primitives and the double compare-and-swap (DCAS) primitive are all encapsulated
3 within one or more functions that implement an LFRCDCAS pointer operation.

1 13. A lock-free implementation of a concurrent shared object comprising:
2 plural component shared objects encoded in dynamically-allocated shared
3 storage; and
4 access operations that, prior to attempting creation or replication of a pointer
5 to any of the component shared objects, increment a corresponding
6 reference count, and upon failure of the attempt, thereafter decrement
7 the corresponding reference count,
8 the access operations decrementing a particular reference count, except when
9 handling a pointer creation failure, no earlier than upon destruction of a
10 pointer to a corresponding one of the component shared objects.

1 14. The lock-free implementation of a concurrent shared object as recited in
2 claim 13, wherein the access operations employ lock-free, reference-count-
3 maintaining pointer operations.

1 15. The lock-free implementation of a concurrent shared object as recited in
2 claim 13, wherein the access operations include one or more of:
3 a lock-free, reference-count-maintaining load operation;
4 a lock-free, reference-count-maintaining store operation;
5 a lock-free, reference-count-maintaining copy operation;
6 a lock-free, reference-count-maintaining destroy operation;
7 a lock-free, reference-count-maintaining compare-and-swap (CAS) operation;
8 and
9 a lock-free, reference-count-maintaining double compare-and-swap (DCAS)
10 operation.

1 16. The lock-free implementation of a concurrent shared object as recited in
2 claim 13, wherein each of the access operations are lock-free.

1 17. The lock-free implementation of a concurrent shared object as recited in
2 claim 13, wherein the access operations employ either or both of a compare-and-swap
3 (CAS) primitive and a double compare-and-swap (DCAS) primitive.

1 18. The lock-free implementation of a concurrent shared object as recited in
2 claim 13, wherein the access operations employ emulations of either or both of the
3 compare-and-swap and double-compare-and-swap operations.

1 19. The lock-free implementation of a concurrent shared object as recited in
2 claim 18, wherein the emulation is based on one of:
3 a load-linked/store-conditional operation pair; and
4 transactional memory.

1 20. The lock-free implementation of a concurrent shared object as recited in
2 claim 13,

3 wherein the incrementing and decrementing are performed using a
4 synchronization primitive.

1 21. The lock-free implementation of a concurrent shared object as recited in
2 claim 13,

3 wherein the concurrent shared object includes a doubly-linked list; and
4 wherein the access operations are performed using a synchronization primitive
5 to mediate concurrent execution thereof.

1 22. A method of transforming an implementation of a concurrent shared data
2 structure from garbage collection- (GC-) dependent to GC-independent form, the
3 method comprising:

4 associating a reference count with each shared object instance;
5 modifying the implementation, if necessary, to ensure cycle-free garbage;
6 replacing pointer accesses in the implementation with corresponding lock-free,
7 reference-count-maintaining counterpart operations; and
8 explicitly managing local pointer variables using a lock-free, reference-count-
9 maintaining destroy operation that frees storage if a corresponding
10 reference count has reached zero.

1 23. The method of claim 22, wherein the replacement of pointer accesses
2 includes one or more of:

3 replacing an access that assigns a shared pointer value to a local pointer
4 variable with a lock-free, reference-count-maintaining load operation;
5 replacing an access that assigns a local pointer value to a shared pointer
6 variable with a lock-free, reference-count-maintaining store operation;
7 and
8 replacing an access that assigns a local pointer value to a local pointer variable
9 with a lock-free, reference-count-maintaining copy operation.

1 24. The method of claim 23, wherein the replacement of pointer accesses
2 further includes:

3 replacing an access that assigns a shared pointer value to a shared pointer
4 variable with:
5 a lock-free, reference-count-maintaining load operation to a local
6 temporary variable;
7 a lock-free, reference-count-maintaining store operation from the local
8 temporary variable; and
9 a lock-free, reference-count-maintaining destroy operation that frees
10 storage associated with the local temporary variable if a
11 corresponding reference count has reached zero.

1 25. The method of claim 22,
2 wherein the lock-free, reference-count-maintaining counterpart operations
3 include object type specific instances thereof.

1 26. The method of claim 22,
2 wherein the lock-free, reference-count-maintaining counterpart operations are
3 generic to plural object types.

1 27. The method of claim 22,
2 wherein the lock-free, reference-count-maintaining destroy operation is
3 recursive.

1 28. The method of claim 22, further comprising:
2 generating a computer program product including a computer readable
3 encoding of the concurrent shared data structure, which is instantiable
4 in dynamically-allocated shared storage, the computer readable
5 encoding further including functional sequences that facilitate access to
6 the concurrent shared data structure and that include the lock-free,
7 reference-count-maintaining counterpart operations.

1 29. A computer program product encoded in at least one computer readable
2 medium, the computer program product comprising:

3 a representation of a shared object that is instantiable as zero or more
4 component objects in dynamically-allocated shared storage of a
5 multiprocessor;
6 at least one instruction sequence executable by respective processors of the
7 multiprocessor, the at least one instruction sequence implementing at
8 least one access operation on the shared object and employing one or
9 more lock-free pointer operations to maintain reference counts for one
10 or more accessed component objects thereof; and
11 the at least one instruction sequence further implementing explicit reclamation
12 of the component objects, thereby freeing storage associated with a
13 particular one of the component objects only once the corresponding
14 reference count indicates that the particular component object is
15 unreferenced.

1 30. The computer program product of claim 29,
2 wherein the zero or more component objects of the shared object are organized
3 as a linked-list; and
4 wherein the at least one access operation supports concurrent access to the
5 linked-list.

1 31. The computer program product of claim 29, at least partially
2 implementing a mutator that provides explicit reclamation of the dynamically-
3 allocated shared storage.

1 32. The computer program product of claim 29, at least partially
2 implementing a garbage collector that reclaims shared storage dynamically-allocated
3 for a mutator and, which employs the shared object in coordination thereof.

1 33. The computer program product of 29,
2 wherein the at least one computer readable medium is selected from the set of
3 a disk, tape or other magnetic, optical, or electronic storage medium
4 and a network, wire line, wireless or other communications medium.

1 34. An apparatus comprising:

2 plural processors;
3 one or more stores addressable by the plural processors;
4 one or more shared pointer variables accessible by each of the plural
5 processors for referencing a shared object encoded in the one or more
6 stores;
7 means for coordinating competing access to the shared object using one or
8 more reference counts and pointer manipulations that employ one or
9 more lock-free pointer operations to ensure that if the number of
10 pointers to the shared object is non-zero, then so too is the
11 corresponding reference count and further that if no pointers reference
12 the shared object, then the corresponding reference count eventually
13 becomes zero.

1 35. The apparatus of claim 34, further comprising:
2 means for freeing the shared object only once the corresponding reference
3 count indicates that the shared object is unreferenced.

RECORDED IN U.S. PATENT AND TRADEMARK OFFICE
RECEIVED MAR 2013 BY USPTO